Reference-distance Eviction and Prefetching for Cache Management in Spark

Tiago B. G. Perez
University of Colorado
Colorado Springs, Colorado
tbarreto@uccs.edu

Xiaobo Zhou
University of Colorado
Colorado Springs, Colorado
xzhou@uccs.edu

Dazhao Cheng
University of North Carolina
Charlotte, North Carolina
dazhao.cheng@uncc.edu

ABSTRACT

Optimizing memory cache usage is vital for performance of in-memory data-parallel frameworks such as Spark. Current data-analytic frameworks utilize the popular Least Recently Used (LRU) policy, which does not take advantage of data dependency information available in the application's directed acyclic graph (DAG). Recent research in dependency-aware caching, notably MemTune and Least Reference Count (LRC), have made important improvements to close this gap. But they do not fully leverage the DAG structure, which imparts information such as the time-spatial distribution of data references across the workflow, to further improve cache hit ratio and application runtime.

In this paper, we propose and develop a new cache management policy, *Most Reference Distance (MRD)* that utilizes DAG information to optimize both eviction and prefetching of data to improve cache management. MRD takes into account the relative stage distance of each data block reference in the application workflow, effectively evicting the furthest and least likely data in the cache to be used, while aggressively prefetching the nearest and most likely data that will be needed, and in doing so, better overlapping computation with I/O time. Our experiments with a Spark implementation, utilizing popular benchmarking workloads show that, MRD has low overhead and improves performance by an average of 53% compared to LRU, and up to 68% and 45% when compared to MemTune and LRC respectively. It works best for I/O-intensive workloads.

CCS CONCEPTS

• **Theory of computation → Caching and paging algorithms**;

KEYWORDS

Cache management, reference distance, Spark, DAG

ACM Reference Format:
Tiago B. G. Perez, Xiaobo Zhou, and Dazhao Cheng. 2018. Reference-distance Eviction and Prefetching for Cache Management in Spark. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225087

1 INTRODUCTION

The emergence of data analytics frameworks [3, 10, 14, 16, 17, 19, 28], which rely on in-memory computing to speed up performance and bypass the hindrance of disk and network I/O, has made the cache management crucial. Even with the significant decrease in the cost of memory over the years that provides an abundance of RAM in modern clusters, the exponential growth of data size for bigdata analytics make cache a precious resource and a bottleneck often.

Caching is a long-established and well studied problem in various computing systems ranging from web servers to operating systems. What differs, and makes data analytics frameworks distinct in this matter, is the availability of information on the data dependency and access pattern before the execution of the application. This data dependency is conveyed through the structure of the directed acyclic graphs (DAG) [23], which is used to organize the workflow of the application. In the popular data-parallel processing framework Spark, this workflow is divided into jobs, stages and tasks which can be exploited for job scheduling [6, 24] and data caching.

Spark uses by default the popular but DAG-oblivious LRU caching policy [13]. Previous studies, namely MemTune [25] and LRC [26] have already made use of the DAG and its data dependency to improve cache management. However, MemTune approach groups Resilient Distributed Datasets (RDDs) into lists and does not sufficiently discretize which and when each RDD will be needed in the cache. For LRC, while it does assign values that differentiate the weight for each data block to be in cache, it does not take into account the impact of data blocks having large gaps in being referenced during the workflow.

In this paper, we ask *how the DAG can be further exploited to improve cache management in Spark?* More specifically, we look in detail how the data dependency is organized along the workflow of the application, and how a new metric *reference distance*, can be used to predict when a data block is needed to be present in the cache, be of low overhead, and be generally applicable to DAG-based in-memory data analytics frameworks.

We propose and develop a novel cache eviction and prefetching policy, *Most Reference Distance (MRD)*, that always evicts the data block whose reference distance is the largest, and prefetches the data blocks whose reference distance is the smallest. Reference distance is defined, for each data block, as the relative distance between the current step in the application execution and the step in the workflow that the data block is needed. The reference distance is initially calculated by parsing the DAG, and later while the application is executed, the reference distance for each block is updated by simply decrementing the value based on the stage ID that is currently executing.

The results based on the testbed implementation with fourteen different benchmark data analytic workloads show that MRD performs very well, but works best for I/O-intensive workloads. Compared to Spark's default LRU caching policy, it reduces application runtime to as low as 20% of original and on average by 53%. MRD improves system performance by up to 68% and 45% when compared to MemTune and LRC, respectively.

The structure of this paper is as follows. Section 2 reviews related work and limitations. Section 3 presents the motivation on our work. Section 4 presents the architecture of MRD. Section 5 is on the performance evaluation and discussion of findings and Section 6 concludes the paper and offers future research directions.

2 RELATED WORK

Least Recently Used (LRU) [13] is a recency-based cache management policy, that keeps track of when each block of data was last accessed, and evicts the one without access for the longest period of time. It assumes that recently accessed data has a higher chance of being needed in the near future. For most of in-memory data analytics systems [10, 16, 19, 28], it is the de facto cache management policy. However, it is oblivious to the data workflow information provided by the DAG, resulting in inefficient and erroneous eviction decisions as shown in the next section.

MemTune [25] dynamically adjusts parameters for memory distribution in Spark. It evicts and prefetches using dependency information from the DAG, but it restricts to local dependencies on runnable tasks, and keeps information of all the required RDD blocks in a series of lists that do not provide the fine-grained time-locality information the DAG is able to provide. While the use of DAG dependencies is an improvement over the oblivious LRU policy, the level of specificity is too general and leaves significant improvements to be made.

Least Reference Count (LRC) [26] traverses the DAG and its dependencies and makes a count of the number of references to each data block. As the application is run, LRC keeps track and updates the count. The intuition being that data blocks that gets referenced the most, are more likely to be needed than its peers, are kept in the cache, and the lowest reference count gets evicted. Our solution also makes use of memory references to data blocks in a critical way. But we make the distinction that reference distance provides a better metric in anticipating blocks that get referenced sooner and those that are further in the future of a workflow, which would keep the reference count high and mislead the necessity of caching a particular block.

DAG-oblivious Caching policies. There are other recently proposed caching systems that do not use DAG information and are orthogonal to our work. Examples include Hyperbolic [5], which utilizes random sampling and priority functions for eviction. Elastic Memory [22] models memory usage and GC, while changing JVM memory limits and reallocation between applications. V-Cache [8] uses a genetic algorithm to allocate caching and machine learning to dynamically resize the cache space. Miniature Simulation [21] utilizes sampling and hashing to model caching behavior and allows the simulation of multiple parameters in caching policies. Favorable Block First (FBF) [11] utilizes the relationships among parity chains to hold the most significant data in buffer cache for partial stripe reconstruction in disk arrays tolerating triple disk failures. It is commonly used in cloud datacenters. MANGO [15] utilizes predictive methods such as fuzzy logic and best-fit algorithms, for memory management in heterogeneous NUMA shared memory architectures with workloads of different priorities.

3 MOTIVATIONS

In this section, we give background information on how Spark's use of the DAG structure can provide insights into data access pattern, how a new metric (reference distance) can be leveraged to improve on the limitations of previous caching policies (LRU, LRC and MemTune), and a motivating example that compares such policies in keeping in the cache the data to be most likely needed again. Although the specifics of this paper are directed at the Spark [28] framework, with some adaptations nothing prevents it from being applicable to other data-parallel frameworks such as Storm [19], Tachyon [10] and Tez [16].

3.1 Data Workflow in Spark

There are two fundamental concepts in understanding how Spark deals with the flow of data when running an application. They are Resilient Distributed Datasets (RDDs) [27] and Directed Acyclic Graphs (DAGs). Whenever a user program is run, the sequence of commands in the code are interpreted, and the creation of RDDs and DAGs as a result represent the transformations, dependencies and workflow between the input, intermediate data and output of that program. This provides an overview of the data access pattern during execution, which can be exploited for improving cache management. An overall picture of RDDs and the DAG can be seen in Figure 1, we briefly breakdown its various components.

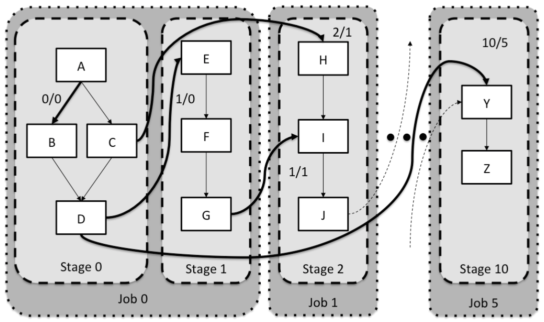
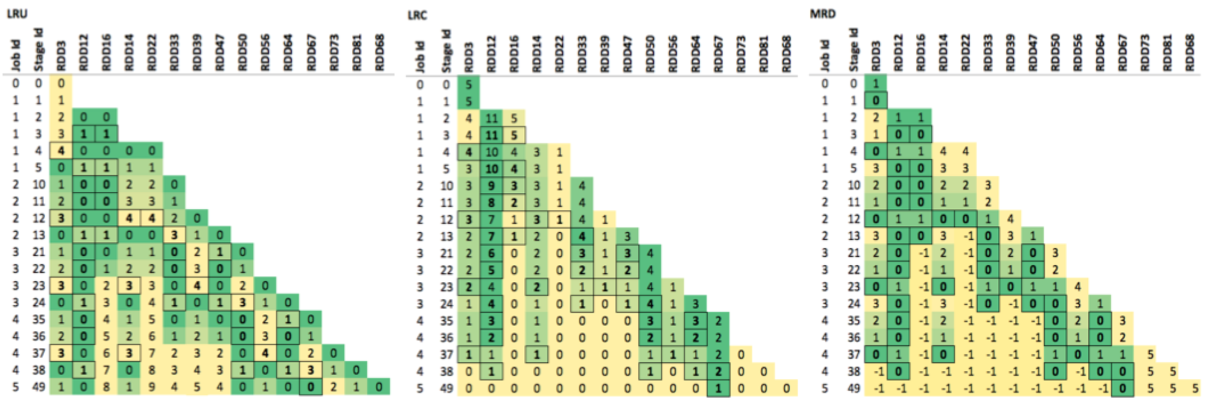


Figure 1: Spark DAG structure, with jobs, stages and data blocks with dependencies, measurements of stage (1st value) and job distances (2nd value).

RDDs are Spark's parallel data structure abstraction, which encapsulates not only the data, but also the transformations that form its lineage, allowing for its reconstruction in case of failure. They can be created from the original input data from a storage system (e.g. HDFS [18], Tachyon [10] and Amazon S3 [2]) or computed



LRU

Job Id	Stage Id	RDD3	RDD12	RDD16	RDD14	RDD22	RDD33	RDD39	RDD47	RDD50	RDD56	RDD64	RDD67	RDD73	RDD81	RDD68
0	0	0														
1	1	1														
1	2	2	0	0												
1	3	3	1	1												
1	4	4	0	0	0	0										
1	5	0	1	1	1	1										
2	10	1	0	0	2	2	0									
2	11	2	0	0	3	3	1									
2	12	3	0	0	4	4	2	0								
2	13	0	1	1	0	0	3	1	0							
3	21	1	0	0	1	1	0	2	1	0						
3	22	2	0	1	2	2	0	3	0	1						
3	23	3	0	2	3	3	0	4	0	2	0					
3	24	0	1	3	0	4	1	0	1	3	1	0				
4	35	1	0	4	1	5	0	1	0	0	2	1	0			
4	36	2	0	5	2	6	1	2	1	0	3	0	1			
4	37	3	0	6	3	7	2	3	2	0	4	0	2	0		
4	38	0	1	7	0	8	3	4	3	1	0	1	3	1	0	
5	49	1	0	8	1	9	4	5	4	0	1	0	0	2	1	0

LRC

Job Id	Stage Id	RDD3	RDD12	RDD16	RDD14	RDD22	RDD33	RDD39	RDD47	RDD50	RDD56	RDD64	RDD67	RDD73	RDD81	RDD68
0	0	5														
1	1	5														
1	2	4	11	5												
1	3	4	11	5												
1	4	4	10	4	3	1										
1	5	3	10	4	3	1										
2	10	3	9	3	3	1	4									
2	11	3	8	2	3	1	4									
2	12	3	7	1	3	1	4	1								
2	13	2	7	1	2	0	4	1	3							
3	21	2	6	0	2	0	3	1	3	4						
3	22	2	5	0	2	0	2	1	2	4						
3	23	2	4	0	2	0	1	1	1	4	1					
3	24	1	4	0	1	0	1	0	1	4	1	3				
4	35	1	3	0	1	0	0	0	0	3	1	3	2			
4	36	1	2	0	1	0	0	0	0	2	1	2	2			
4	37	1	1	0	1	0	0	0	0	1	1	1	2	0		
4	38	0	1	0	0	0	0	0	0	1	0	1	2	0	0	
5	49	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

MRD

Job Id	Stage Id	RDD3	RDD12	RDD16	RDD14	RDD22	RDD33	RDD39	RDD47	RDD50	RDD56	RDD64	RDD67	RDD73	RDD81	RDD68
0	0	1														
1	1	0														
1	2	2	1	1												
1	3	1	0	0												
1	4	0	1	1	4	4										
1	5	3	0	0	3	3										
2	10	2	0	0	2	2	3									
2	11	1	0	0	1	1	2									
2	12	0	1	1	0	0	1	4								
2	13	3	0	0	3	-1	0	3	1							
3	21	2	0	-1	2	-1	0	2	0	3						
3	22	1	0	-1	1	-1	0	1	0	2						
3	23	0	1	-1	0	-1	1	0	1	1	4					
3	24	3	0	-1	3	-1	0	-1	0	0	3	1				
4	35	2	0	-1	2	-1	-1	-1	-1	0	2	0	3			
4	36	1	0	-1	1	-1	-1	-1	-1	0	1	0	2			
4	37	0	1	-1	0	-1	-1	-1	-1	1	0	1	1	5		
4	38	-1	0	-1	-1	-1	-1	-1	-1	0	-1	0	0	5	5	
5	49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	5	5	5

Figure 2: Comparison of caching policies (LRU, LRC and MRD) for ConnectedComponents (CC) workload.

from a previous RDD. They are represented in Figure 1 by the lettered boxes (A-Z) and are partitioned into several blocks that are distributed across the machines in the cluster. The lines connecting the boxes show the data dependency among the RDDs, these can cross the boundaries of stages and jobs.

The DAG, which is the overall workflow of the application, is built from the RDDs' dependancies. Whenever a user performs an action on an RDD (e.g. `RDD.count`), this splits the user program into jobs and triggers job execution. Whenever a user performs a narrow transformation on an RDD (e.g. `RDD.map()`), each block of a parent RDD gets transformed into the next child RDD, allowing it to be pipelined and the data transformation stays within a stage. Whenever a user performs a wide transformation on an RDD (e.g. `RDD.goupByKey()`), it requires a shuffle from all the parent RDDs into the next child RDD, causing a split between stages.

Hence having access to the DAG and its sequence of RDDs transformations, allows for a semi-omniscient view of data access. In fact the `DAGScheduler` component of Spark uses a depth-first search to traverse the job DAG and select and submit runnable tasks (i.e. for which all parent RDDs have already been computed) to the `TaskScheduler`. However, we don't have a fully-omniscient view of data access. Since we don't know exactly the order the tasks are going to be run, we thus only approximate Belady's MIN [4] optimal caching policy.

3.2 Reference Distance

With the revelation of the data access workflow provided by the DAG structure, which is easily retrieved from the `DAGScheduler`, we devise a new metric which can leverage the time-locality information contained within. We call it *reference distance*. It is a locality measurement in the sense that for each piece of data block we are measuring the relative distance between the current point in the DAG that the program is executing, and the next time that data block will be referenced (i.e. when the parent RDD will be accessed from memory or disk to create the next child RDD). It is a time measurement, in the sense that even though we won't know precisely how long in seconds or minutes the reference will happen, when a reference distance is compared to another, we can get a sense of immediacy for the next data block access. This gives a high level of predictability in guessing which data blocks are most likely to be needed in the cache sooner.

For our work, we refer to two types of reference distances, they are stage distance and job distance. Referring back to Figure 1 we give examples for the measurement of both. The stage distance, which is the first value on the bold lines of the figure, are measured relative to Spark's stages within a DAG. These are sequentially numbered and contained in the variable `StageID`, making it easy to simply subtract the current stage the application execution is, from the stage on which a particular data block will need to be accessed. The job distance, which is the second value on the bold lines of the figure, is correspondingly relative to Spark's jobs within a DAG. These are similarly sequentially numbered and contained in the variable `JobID`. As seen in Figure 1, block **D** can have both job distance 5, and stage distance 10, although both reference distances have good predictive capabilities, the finer grained stage distance offers a better metric, as shown by experiments in Section 5.7.

3.3 Eviction Probability Across a Workflow

In Figure 2, we make a comparison of three caching policies (LRU, LRC and MRD) behavior in the ConnectedComponents (CC) workload, and show how the probability of caching and eviction changes as the workload is run. On the top of each graph we have the RDDs that are cached by the application. On the left the JobID and StageID shows points of reference for the application workflow. The numbers in bold with border outlines show when an RDD is referenced (data dependencies) during the application run. The color varies from green (most likely to be in cache) to yellow (most likely to be evicted) for each particular policy. The numbers themselves reflect the respective policies' metric towards the data. For LRU the value for each RDD is zero at creation and gets increased the further away since its last reference, being reset to zero after each reference (higher values get evicted). For LRC the value starts at the maximum number of references and gets decreased after each reference (lowest values get evicted). For MRD the values indicate how many stages away from getting referenced, being reset to the new distance after each reference. After there are no more references, the value defaults to a maximum value, while the highest values gets evicted.

LRU does correctly maintain necessary data cached at several moments, especially when it is reference often. But RDDs that have gaps in their references suffer in favor of recently referenced ones. In Figure 2 such is the case for RDD3 in stage 4, and RDD14 and 22 in stage 12 and so on, that get evicted in favor of maintaining recently referenced data blocks that don't have the same level of urgency.

LRC improves upon LRU by keeping in cache the needed data blocks more often, due to future references in the workflow RDD3 in stage 4 and RDD14 in stage 12 have a higher probability to escape eviction. But, it still fails in cases like RDD 22 in stage 12, where its single reference in the entire workflow places it at a disadvantage to other RDDs that have future references, and as such keep a high reference count, misleading the LRC policy in favoring its peers and not RDD22.

MRD improves upon LRC by taking into consideration the reference distance for each data block. This resolves the issues encountered for both LRU and LRC described previously. Giving the highest probability of keeping in cache those data blocks that will be reference next, as can be seen in Figure 2. In the case of prefetching, MRD will bring to cache the data block with the lowest value that is not in cache yet. Keep in mind that the exact data in cache will depend on the size of the RDD's data blocks and the size of the cache available. As such, data blocks with the same reference distance might not all fit the cache, a methodology to prioritize which data block is cached in case of such ties are left for future work.

4 SYSTEM DESIGN

In this section, we present the design of the Most Reference Distance (MRD) cache management policy, which makes the cache eviction and prefetching decisions based on the stage reference distance extracted from the DAG, and its implementation in Spark.

4.1 Most Reference Distance (MRD)

We first define reference distance.

Definition 1 (Reference Distance). *For each data block, the reference distance is define as the relative distance between the current step in the application execution and the step in the workflow that the data block has a dependent, utilizing a workflow-based subdivision such as jobID or stageID.*

The Most Reference Distance (MRD) policy will always evict the data block whose reference distance is the largest, and prefetches the data blocks whose reference distance is the smallest. Referring back to Figure 1, block **D** can have both stage distances 1 and 10. MRD will keep track of the distance values for all the references, but for comparison it will only use the lowest one. As the application execution moves beyond a point where there is a reference, that value is deleted, and the next lowest one is used. In case there are no more distance values recorded, MRD assumes that the data block is no longer referenced, and an infinite distance (negative value) is used.

Reference distance allows for MRD to quickly detect and dispose of inactive data with infinite distances, thus clearing space in the cache. It allows for an easy way to compare data that will soon be reference among its peers for prefetching, and avoids the inaccuracy of maintaining data cached, that is still active, but will only be needed much later in the application workflow, and hence have large reference distances, making it second in line to inactive data to be considered for eviction, and the last to be considered for prefetching.

In our preliminary study, we considered two popular benchmark suites, SparkBench [12] and HiBench [9] suites. But as seen in Table 1, our preliminary study found that the measured job distances and stage distances for HiBench were much smaller due to the nature of the workload or its particular implementation, and hence offered less opportunities for MRD to exploit the DAG. HiBench workloads were dropped from the final experiments.

Table 1: Reference distance characteristics of benchmark workloads.

Workload	Average Job Distance	Maximum Job Distance	Average Stage Distance	Maximum Stage Distance
SparkBench Suite				
K-Means (KM)	5.15	16	5.34	19
Linear Regression (LinR)	1.24	5	1.76	8
Logistic Regression (LogR)	1.53	6	2.00	9
SVM	1.48	6	1.96	10
Decision Tree (DT)	2.71	9	4.38	15
Matrix Factorization (MF)	1.56	7	3.31	18
Page Rank (PR)	1.74	5	6.08	19
Triangle Count (TC)	0.07	1	1.23	6
Shortest Paths (SP)	0.19	1	1.19	4
Label Propagation (LP)	7.19	22	28.37	85
SVD++	3.51	11	6.82	23
ConnectedComponent (CC)	1.30	4	5.31	16
StronglyConnectedComponent (SCC)	7.77	24	29.96	90
PregelOperation (PO)	1.28	4	5.45	16
HiBench Suite				
Sort	0.00	0	0.00	0
WordCount	0.00	0	0.00	0
TeraSort	0.22	1	0.22	1
PageRank	0.00	0	0.09	2
Bayes	2.09	7	3.23	9
K-Means	6.08	19	6.60	25

Ideally, to have an accurate value for every data block's reference distance, we need to view the entire application's DAG. Unfortunately in systems like Spark and Tez, the DAG is broken down in parts and only available per job submission, hence applications with several jobs will provide their information in fragments. This leaves MRD with two modus operandi as follows.

First, since the study of production traces show that a high percentage of workloads running in a cluster are recurring applications [7], meaning they are periodically re-run with just new data as input, we save the DAG profile of the application from previous runs, in essence storing the reference distance information for each RDD.

Second, for the non-recurring applications, within a job with several stages, it still is possible to compute a stage distance relative to data blocks reference within the same job. Data blocks without references within the current jobs are assumed to have infinite distances until a new job is submitted, and with the DAG for the subsequent job, MRD can update the reference distances. Obviously this approach is meaningless for job distances since they will always be either infinite or zero, hence another reason why stage distances are more fine-grained and preferred. We provide performance comparisons of the two methods in Section 5.8.

4.2 Implementation

Architecture overview. Figure 3 gives the overall architecture of MRD where shaded components represent the major implementations. MRD has two main centralized components, `AppProfiler` and `MRDmanager`, and a distributed component, `CacheMonitor` that is implemented in each worker node. Most of the modifications to Spark's components were in: `DAGScheduler, BlockManagerMaster, BlockManagerMasterEndpoint, BlockManagerSlaveEndpoint, BlockManager`, and `MemoryStore`. The main APIs for the MRD implementation can be seen on Table 2.

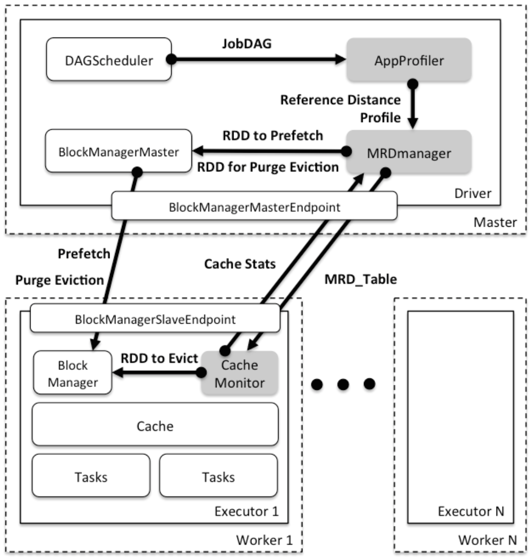


Figure 3: MRD system architecture.

AppProfiler component. Performs two tasks, for the first time of a recurring application or a ad-hoc non-recurring application is run it receives the job DAG from the `DAGScheduler` for parsing, identifies and calculates the initial reference distance based on the DAG and passes it to the `MRDmanager`, as the application is executed and further job DAGs are received, the reference distance is processed and passed on in a similar fashion, this in essence creates an application profile that is stored in the `AppProfiler` for future use. In the case of identifying a recurring application, the `AppProfiler` instead can send the entire application DAG to the `MRDmanager`.

MRDmanager component. Implements the main logic of the MRD policy. It receives the initial and timely updates for the reference distances from the `AppProfiler`, but is the actual component that will keep track of what stage the execution is and decrement the reference distances as the application is run, maintaining and updating the current distance profile. It also runs part of the MRD eviction and prefetching algorithm that is explained in the next subsection. Passes on the prefetching orders or gives the all-out-purging eviction order in case an RDD is no longer needed for the `BlockManagerMaster` to act upon across the cluster, and updates the `CacheMonitor` in each node with the current reference

Table 2: Key APIs for Spark implementation.

API	Description
parseDAG	The `AppProfiler` parses the DAG information received from the `DAGScheduler` and creates the reference distance profile for the job or application
updateReferenceDistance	The `MRDmanager` updates the reference distance profile with new values received from the `AppProfiler`
newReferenceDistance	The `MRDmanager` updates the reference distance values in the entire profile for each new stage the application execution proceeds to
sendReferenceDistance	The `MRDmanager` sends the required RDD data block reference distances to each `CacheMonitor` in the cluster
getReferenceDistance	The `CacheMonitor` requests the reference distance profile for new RDD blocks from the `MRDmanager`
reportCacheStatus	`CacheMonitor` reports information like memory availability, and hit ratios to the `MRDmanager` periodically
evictBlock	When the cache is full, the `CacheMonitor` send the block to evict to `BlockManager` based on the reference distance profile it has
prefetchBlock	When there is space available, the `MRDmanager` sends the block to prefetch to the `BlockManager`

distance to make local decisions in case of memory pressure (such as a prefetch) forces an eviction.

CacheMonitor component. Is deployed across the cluster, in each worker node. It receives reference distance updates from the `MRDmanager` for local eviction decisions, carries out the prefetching orders given by `MRDmanager` and sends back information on hit ratio and memory status information to the `MRDmanager` to make prefetch decisions. This communication is made through the `BlockManagerMasterEndpoint` and `BlockManagerSlaveEndpoint`.

Setup workflow. When an application is submitted, Spark launches on the master node the Spark driver as well as a `SparkContext` object. In turn, the `SparkContext` instantiates the `AppProfiler` and `MRDmanager` components as well as the usual Spark components: `DAGScheduler`, `BlockManagerMaster` and `BlockManagerMasterEndpoint`. Next, the Spark driver launches across its worker nodes in the cluster, the executor components, which includes our `CacheMonitor` as well as the usual Spark components `BlockManager` and `BlockManagerSlaveEndpoint`.

Default workflow. When the application starts its run, the cache has ample vacancy, and as new RDDs are created and its data blocks are cached based on the user defined program, it will eventually become full. At some point in the workflow, an already created RDD is needed, in that case it will either use the already cached data blocks, or fetch from its local disk or a remote node the needed data blocks. If the cache does not have enough space, it follows its default LRU caching policy and evict the longest idle data block.

MRD eviction workflow. Overrides the default behavior in two ways, first `CacheMonitor` will choose a data block that has the greatest reference distance value to evict whenever it encounters pressure to free up space, whether its the data blocks from a new RDD being cached, or an already existing RDD being prefetched. Second, from time to time, when `MRDmanager` detects that an RDD

Algorithm 1 MRD eviction and prefetching.

```
1: Input: N, S, free_memory, threshold
2: /*Initial Phase*/
3: for each job $j$ of $J$ do
4:     $MRD\_Table \leftarrow$ reference distances from DAG
5:     for each node $n$ of $N$ do
6:         send($MRD\_Table$)
7:     end for
8: end for
9: for each stage $s$ of $S$ do
10:    update($MRD\_Table$)
11: end for
12: /*Eviction Phase*/
13: if data block $d_i < 0$ AND $d_i \subset MRD\_Table$ then
14:     for each node $n$ of $N$ do
15:         evict($d_i$)
16:     end for
17: end if
18: while new data block sizeof($d_n$) $> free\_memory$ do
19:     $d_e \leftarrow$ highest($MRD\_Table$)
20:     evict($d_e$)
21: end while
22: cache($d_n$)
23: /*Prefetching Phase*/
24: for each node $n$ of $N$ do
25:     $d_p \leftarrow$ lowest($MRD\_Table$)
26:     if sizeof($d_p$) $< free\_memory$ OR $free\_memory > threshold$ then
27:         prefetch($d_p$)
28:     end if
29: end for
```

is no longer needed (all of its data blocks have reference distance to infinite), a cluster wide purge order is sent to all CacheMonitors, preemptively evicting and freeing up cache space, instead of waiting for memory pressure to set it off. This aggressive behavior allows the cache to free space more frequently instead of only doing eviction when there isn't enough memory for the next block.

MRD prefetching workflow. Overrides the default behavior by MRDmanager preemptively fetching data blocks that have the lowest reference distance (i.e. most likely to be needed next) when a certain threshold of memory is free (which might cause memory pressure to evict the largest reference distance block as described above), or if the next data block fits into the current memory if the threshold has not been met. This aggressive behavior allows the most needed blocks to be placed in memory earlier, overlapping the stalling time of I/O with computation.

4.3 Eviction and Prefetching

The pseudocode for the eviction and prefetching algorithm in MRD can be seen in Algorithm 1. Parts of it run in the centralized MRDmanager component, while others execute in the distributed CacheMonitor.

Initial phase workflow. Lines 3 to 11, MRDmanager adds the new reference distances to the *MRD_Table* on a per job basis in case of ad-hoc applications, and updates the values in case of a discrepancy for recurring applications, then the various work nodes in the cluster are updated. Also, for each stage that is processed, the MRD_Table is updated with the new distance at the master and nodes, usually a decrement of one for each distance, unless some stages are skipped, regardless the appropriate value is calculated based on the StageID.

Eviction phase workflow. Lines 13 to 22, MRD performs evictions in two instances. First, for the MRDmanager in case there exists a data block with infinite reference distance (negative value), and is contained in the MRD_Table, it will evict that particular block from every worker node in the cluster. Second, for the CacheMonitors in case a new block is being allocated space in the cache, and the size of that block is greater then the available space, then the data blocks with the highest reference distance values are evicted until there is enough space available to cache the new block.

Prefetching phase workflow. Lines 24 to 29, MRDmanager performs prefetching by selecting the data block with the lowest (non-negative) value in the MRD_Table, and respecting data locality, checks for each worker node if the data block would fit in memory (a certain prefetch), or if there is sufficient memory available to force a prefetch (also forcing an eviction by the CacheMonitor). The threshold value is set experimentally at 25% of the cache space.

4.4 Overheads and Fault Tolerance

Storage and computation overhead. MRD book-keeping is relatively small and comparable to the LRU (default) caching policy. The largest MRD_Table, measured in KBs contained less then 300 references. In terms of computations, only a small sorting is necessary among the few references, and leads to undetectable differences in CPU processing.

Communication overhead. In order to keep the communication overhead to a minimum, each CacheMonitor worker node has a copy of the reference distance profile, and can request and receive updates as necessary from the MRDmanager. These occur in the case a new job DAG is processed for ad-hoc applications, or a discrepancy in the application DAG is detected. Also, from time to time, as an RDD reference distance reaches infinity, the all-out purge eviction order is triggered by the MRDmanager.

Prefetching overhead. Although we acknowledge that there is a possibility of encountering a big enough data block, whose size would take up almost the entire worker node cache, and hence the prefetching would cause the eviction of other data blocks with lower reference distances. Such cases would be counter productive, and lead to performance degradation, but we believe this is a rare case. Improvements where the soon to be pre-fetched data block reference distance is checked against the currently cached blocks are left for future work, but we believe the overhead of having to do a pre-check for every pre-fetching outweighs the benefits of our aggressive pre-fetching policy.

Fault tolerance. In the event of worker node failures, and the loss of local reference distance profiles, the MRDmanager re-issues a copy of the MRD_Table to the new nodes. For recurring applications that might have not completed in their first run, the AppProfiler resumes the creation of reference distance profile with subsequent runs, in the same fashion that it checks for discrepancies during future runs.

5 EVALUATION

In this section, we demonstrate the efficacy of MRD with fourteen different workloads in three cluster settings. We first provide the overall best performance acquired when compared to the default LRU cache policy, then compare specific workloads to the results of

Table 3: SparkBench benchmark characteristics.

Workload	Category	Data Input Size	Total Stage Inputs	Total Shuffle R/W	Characteristics Jobs/Stages/Active Stages/RDDs/ References per RDD/References per Stage	Job Type
K-Means (KM)	Machine Learning	5.5 G	59.3 G	64.5K/64.5K	17 / 20 / 20 / 37 / 5.57 / 1.95	Mixed
Linear Regression (LinR)	Other Workloads	7.7 G	35.8 G	19.8K/19.8K	6 / 9 / 9 / 24 / 5.00 / 0.56	CPU intensive
Logistic Regression (LogR)	Machine Learning	11.1 G	42.7 G	59.1K/59.1K	7 / 10 / 10 / 25 / 6.00 / 0.60	CPU intensive
SVM	Machine Learning	3.8 G	19.1 G	3.2G/3.1G	10 / 28 / 17 / 40 / 3.50 / 0.41	CPU intensive
Decision Tree (DT)	Other Workloads	3.5 G	30.4 G	5.3M/5.3M	10 / 16 / 16 / 29 / 4.00 / 0.25	CPU intensive
Matrix Factorization (MF)	Machine Learning	1.1 G	9.4 G	1.9G/1.9G	8 / 64 / 22 / 103 / 3.11 / 1.27	Mixed
Page Rank (PR)	Web Search	934 M	12.7 G	121M/118M	7 / 69 / 21 / 95 / 2.27 / 2.38	I/O intensive
Triangle Count (TC)	Graph Computation	268 M	3.7G	9.4G/9.2G	2 / 11 / 11 / 74 / 0.80 / 0.73	Mixed
Shortest Paths (SP)	Other Workloads	2.9 G	9.6G	125M/125M	3 / 8 / 7 / 34 / 1.33 / 1.14	Mixed
Label Propagation (LP)	Other Workloads	1.3 M	333M	6.2M/3M	23 / 858 / 87 / 377 / 4.09 / 3.06	I/O intensive
SVD++	Graph Computation	453 M	22.9G	9.4G/9.4G	14 / 103 / 27 / 105 / 3.32 / 2.33	I/O intensive
ConnectedComponent (CC)	Other Workloads	2.4 G	17.2G	0.7G/0.6G	6 / 50 / 19 / 85 / 2.87 / 2.26	I/O intensive
StronglyConnectedComponent (SCC)	Other Workloads	81 M	6.2G	121.3M/113.7M	26 / 839 / 93 / 560 / 4.22 / 3.54	I/O intensive
PregelOperation (PO)	Other Workloads	1.4 G	38.4G	0.8G/0.8G	17 / 467 / 65 / 283 / 3.55 / 3.25	I/O intensive

two previously published caching management solutions. Then we perform specific testing to show the behavior of MRD under several scenarios, such as different cache sizes, use of job distance instead of stage distance, ad-hoc versus recurring runs and the impact of varying the number of iterations of a workload.

5.1 Workloads

For this research we use the widely known and used SparkBench [12] suite. To test MRD performance, we adopt a mix of machine learning, graph computation and other types of applications for a total of fourteen SparkBench workloads. They are representative of many data analytics jobs and offer a good variety of different properties. The characteristics of each workload are given in Table 3.

5.2 Cluster Configurations

Our testbed was composed of various virtualized machine configurations, always with one as master and the rest slaves, with the specifics given on Table 4. Our main set of tests were executed in our university virtual environment with 25-nodes. We also configured our cluster settings to emulate two different environments. A 20-node AmazonEC2 [1] *m4.large* for a comparison to the LRC work and the 6-node System G [20] for a comparison to the MemTune work. The rest of the configuration for each machine is the same with 200GB of disk space, running Linux Ubuntu 15.10, with Spark 2.0.0 standalone mode, and the underlying HDFS [18] with Hadoop 2.6.4 with block size of 128MB. When required by our experiments we altered the `spark.memory.fraction` and `spark.executor.memory` parameters of Spark to simulate different cache sizes, otherwise default parameters are used.

Table 4: Cluster environments.

Cluster Setup	VMs	vCPU	RAM	Network	Equivalency
Main cluster	25	4	8 GB	500 Mbps	N/a
LRC cluster	20	2	8 GB	450 Mbps	Amazon EC2 m4.large
MemTune cluster	6	8	8 GB	1 Gbps	System G

5.3 Overall Performance of MRD

In our *Main cluster* we executed each workload with several cache sizes and averaged out the results out of 20 runs, the results for the best overall performance gain for each workload-cache combination is shown on Figure 4, compared to the normalized Job Completion Time (JCT) of the default LRU in Spark. We show the results for three different scenarios: MRD with eviction-only, MRD with prefetch-only and MRD with both eviction and prefetching enabled. In general MRD performs very well, with significant decreases in application runtime and increase in cache hit ratio.

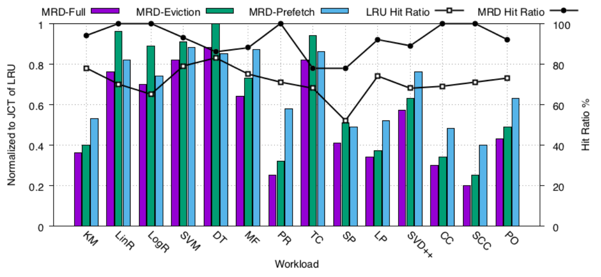


Figure 4: MRD best performance out of several cache sizes.

Eviction-only Results. The application runtime values were reduced on average to 62% of the original JCT. The values reached as low as 25% for the StronglyConnectedComponent (SCC) workload, and as high as 100% (in other word, having no effect) for the DecisionTree (DT) workload. Showing that in general the eviction policy provides the bulk of the improvement for MRD.

Prefetch-only Results. The application runtime values were reduced on average to 67% of the original JCT. The values reached as low as 40% for the StronglyConnectedComponent (SCC) workload, and as high as high as 88% for SVM and DecisionTree (DT). Showing that while the aggressive prefetching on average is less effective, it had better results then pure eviction for some workloads.

Full MRD Results. The application runtime values were reduced on average to 53% of the original JCT. The values reached as low as 20% for the StronglyConnectedComponent (SCC) workload, and as high as high as 88% for DecisionTree (DT). Our best results have come from workloads that are I/O intensive, but we also believe is due to its characteristics of having both high values for both stage reference distance and average references per stage, giving MRD many opportunities to evict the least required data blocks.

Cache hit ratio. For readability we only show the cache hit ratio for LRU and the full implementation of MRD, as is expected, the hit ratio for all workloads have increased, showing that MRD is a better predictor of needed data blocks then the default LRU. The effects on performance vary as there is no linear relationship between the increases of hit ratio and performance, where workload characteristics such as RDD sizes and network I/O come into play. Not all workloads achieve a 100% cache hit ratio, for some experiments the cache available is not able to accommodate pre-fetching of the most needed data block.

5.4 Comparison to LRC

We conducted further tests in our *LRC cluster* to compare results of LRC [26] to the MRD policy. To make the comparison as fair as possible, we emulate their environment and match workload characteristics. Taking the best values from their experiments and ours, we can see in Figure 5 that MRD improves performance up to 45% for ConnectedComponents (CC), but also that overall, the results for MRD were better by an average of 30%. This is in line with expected results as reference distances provides a better prediction metric then reference count, and avoids the pitfall of RDDs that remain in cache due to high count values, but only get referenced further in the future.

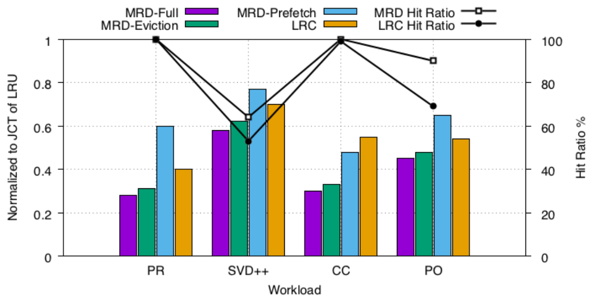


Figure 5: Comparison to LRC policy.

5.5 Comparison to MemTune

We conducted further tests in our *MemTune cluster* to compare results of MemTune [25] to the MRD policy. To make the comparison as fair as possible, we emulate their environment and match workload characteristics. Taking the best values from their experiments and ours, we can see in Figure 6 that MRD improves performance up to 68% for PageRank (PR), but also that overall, the results for MRD were better by an average of 33%. This shows that MRD is able to capture with a finer granularity the details from the DAG in order to predict which data blocks are needed. Only LogisticRegression (LogR) shows a slight disadvantage in performance, this is in line with workloads that have low values for reference distances, where MRD is not fully capable of improving cache performance.

5.6 Impact of Different Cache Sizes

On previous experiments, we displayed the best results for each workload, in Figure 7 we provide more details on the effects of cache size in the performance of MRD in the *LRC cluster* for the SVD++ workload, with further comparisons to the LRC policy. As is expected, the smaller the cache size, the lower the cache hit ratio and longer the application runtime. However, regardless of the cache size, MRD policy outperforms both the LRU and LRC policies.

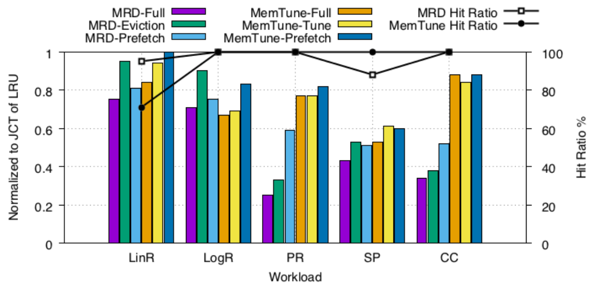


Figure 6: Comparison to MemTune policy.

In a similar fashion to LRC, MRD also has the added benefits of cache space savings. With a much smaller cache size, MRD is able to match the hit ratio of LRU. For example, to achieve a target hit ratio of 68% for SVD++, LRU requires 0.88 GB of cache space. In comparison, MRD requires only 0.33 GB, the equivalent of 63% savings in cache space. Other workloads present similar cache space savings. This is significant as it leads to resource and cost savings.

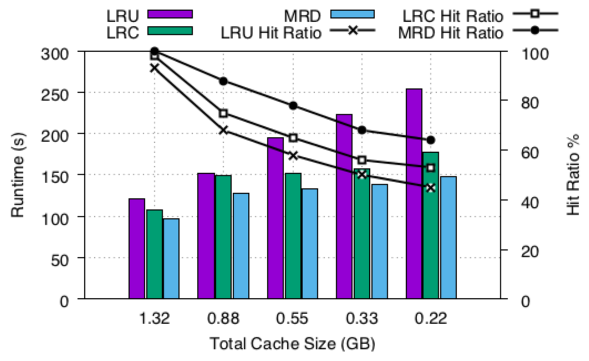


Figure 7: Effects of cache size on hit ratio and runtime, with 3 policies for the SVD++ workload.

5.7 Impact of Stage Distance vs. Job Distance

As described in Section 3.2, our use of reference distance had two possible metrics, job distance and stage distance. Intuitively and confirmed through testing, stage distance provides a better and more fine-grained metric for the MRD policy. In this experiment we show in more detail the impact of using the job distance. In Figure 8 we can see how the use of job distance significantly degrades both the performance and cache hit ratio of LabelPropagation (LP) which has a large ratio (3.17) of active stages (87) to jobs (23). Since all the references within the same job are treated as equals for job distance, the predictive power of MRD is significantly diminished when a job contains several stages and references, by splitting the workflow in a finer-grained stage distance this problem is overcome. While for K-Means (KM) which has a low ratio (1.18) of active stages (20) to jobs (17), the metric used has almost no discernible difference in terms of hit ratio and performance, since in this case stages and jobs are almost equivalent.

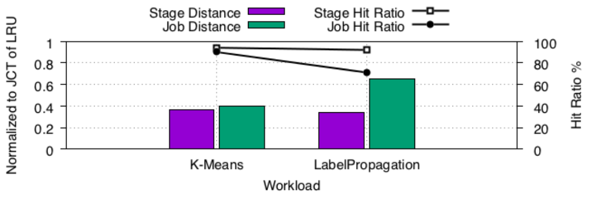


Figure 8: Effects of reference distance metrics.

5.8 Impact of Ad-Hoc vs. Recurring Runs

As described in Section 4.1, our solution deals with limitations in viewing the entire application DAG in two ways. For first-runs and ad-hoc applications, it will build the reference distance profile one-job-at-a-time, for recurring applications that have already been profiled, MRD benefits from seeing the entire application DAG from the start of the run. In this experiment we show in more detail the impact of having the entire DAG on performance and cache hit ratio in Figure 9. For K-Means (KM) where the application is broken down into 17 jobs and has on average 5.57 references per RDD along the entire workflow, the lack of an application-wide DAG view is detrimental. This is because, on the lack of better information, MRD will assume that reference distances to be infinite across job boundaries, hence RDDs with future reference will erroneously be considered good candidates for eviction and poor candidates for prefetching. While for TriangleCount (TC) which has only 2 jobs, and on average 0.8 references per RDD along the entire workflow, the results show that the impacts on performance and cache hit ratio are indiscernible. Particularly in this case, the overall low performance of TriangleCount (TC) with recurring runs is due to its workload characteristic of low average references per RDD, which limits the significance of improving performance with evictions and prefetching.

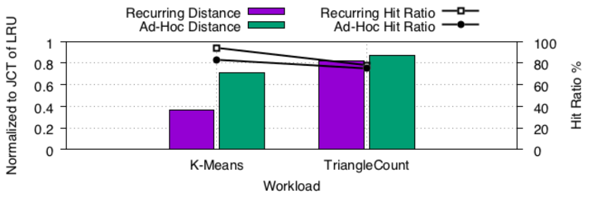


Figure 9: Effects of DAG information availability.

5.9 Impact of Iterations in Workloads

All previous experiments were conducted with the default configurations of SparkBench, however one particular workload parameter of interest to the MRD policy regards the number of iterations. This is important, since increasing iterations, there is a tradeoff between increasing the accuracy of the solution the workload is calculating, with an increase in the time it takes for the application to complete its run. This is an option that the application's user might be willing, or even need to make. From the MRD policy standpoint, the significance is that the increase in iterations usually implies in the increase in number of jobs, stages and cache references an application makes. As a consequence, MRD is able to better leverage evictions and prefetches, and improves the cache hit ratio and performance.

For this experiment we tripled the number of iterations of the workloads that possessed the parameter, on average the number of jobs increased by 59% and the number of stages by 78%, one noticeable mention is the DecisionTree (DT) workload, where there was no impact on either. As can be seen in Figure 10, there is an improvement in performance, decreasing the average JCT from 62% to 54% and increasing the hit ratio from 94% to 96%. However, the effects are not the same across all workloads, and do not scale by simply increasing the number of iterations indefinitely. This is because the increase in jobs and stages is not equal across the workflow, affecting only a part of the application. As such, increasing the number of iterations suffers from the effects of diminishing returns.

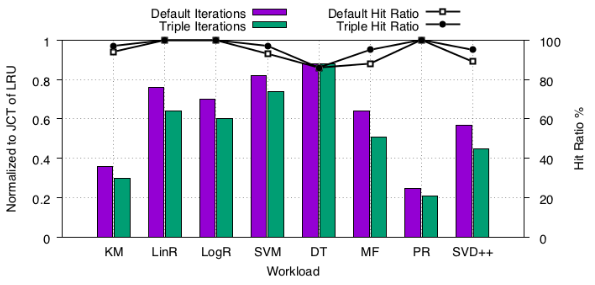


Figure 10: Effects of iterations in workload.

5.10 Comparisons and Discussions

When analyzing the series of experimental results, we find out that three workload characteristics have significant affect on the performance and cache hit ratio for the MRD policy. These workloads are I/O-intensive, high stage distance and high reference per stage workloads.

I/O-intensive workloads. Workloads such as: PageRank, SVD ++, ConnectedComponents (CC) and PregelOperation (PO) are all I/O intensive. By properly caching the most needed data blocks, allows the MRD caching policy the best possibilities to overlap network and disk I/O with computation time, and hence achieve significant performance improvements over the default LRU caching policy.

High stage distance workloads. Figure 11 plots the performance of the various workloads against their average stage distances. We observe that there is a tendency of low stage distance workloads such as SVM achieving low reductions in their JCTs, while high stage distance workloads such as LabelPropagation (LP) have significant improvements to their JCTs. This is because higher average stage distances implies bigger gaps between each reference to an RDD, with such gaps occurring frequently throughout a workload, it allows the MRD policy to perform various evictions and prefetches that improve cache hit ratio and performance.

High reference per stage workloads. Figure 12 plots the performance of the various workloads against their average references per stage. We observe there is a tendency of low reference per stage workloads such as TriangleCount (TC) achieving low reductions in their JCTs, while high references per stage workloads such as K-Means (KM) have significant improvements to their JCTs. This is because higher average references per stage implies more competition between data blocks to occupy the limited cache space, with such bottlenecks occurring frequently throughout a workload, it becomes more critical to evict the proper idle data block in favor of a needed one, it allows the MRD policy to improve on the cache hit ratio and thus performance.

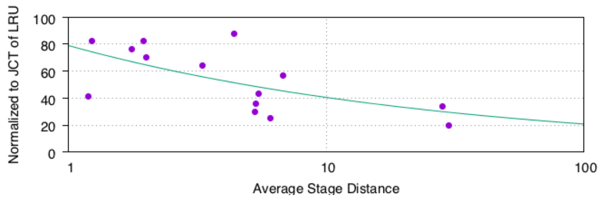


Figure 11: Relationship of performance and stage distance, with trendline of R^2=0.46.

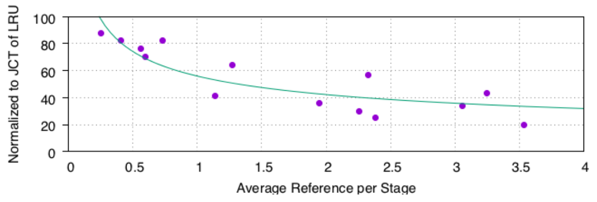


Figure 12: Relationship of performance and average references per stage, with trendline of R^2=0.71.

6 CONCLUSION & FUTURE WORK

In this paper, we design a reference-distance based cache management policy, Most Reference Distance (MRD), which evicts and prefetches data blocks with the greatest and smallest reference distance respectively. Reference distance is measured as the stage distance of the current stage to the stage a data block is referenced. We implemented MRD on Spark. Where experimental results show a reduction in the application runtime to as low as 20% of the original value when compared to Spark's default LRU and on average by 53%. It outperformed previous work, improving performance by up to 68% and 45% when compared to MemTune and LRC respectively. It works best for workloads characterized as I/O-intensive, and workloads that have high stage distance and high reference per stage values.

Our future work includes expanding MRD to other frameworks, testing with more benchmarks, cluster configurations and modifying the prefetching memory threshold to be dynamic and automated.

ACKNOWLEDGMENTS

This research was supported in part by U.S. NSF grant CNS-1422119 and CAPES scholarship BEX 13424-13-0.

REFERENCES

[1] AmazonEC2. https://aws.amazon.com/ec2/.
[2] AmazonS3. https://aws.amazon.com/s3/.
[3] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated memory caching for parallel jobs. In *Proc. USENIX NSDI.*
[4] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.
[5] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. 2017. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC.*
[6] Dazhao Cheng, Yuan Chen, Xiaobo Zhou, Daniel Gmach, and Dejan Milojicic. 2017. Adaptive scheduling of parallel jobs in spark streaming. In *Proc. of IEEE INFOCOM.*
[7] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proc. ACM EuroSys.*
[8] Yanfei Guo, Palden Lama, Jia Rao, and Xiaobo Zhou. 2013. V-cache: Towards flexible resource provisioning for multi-tier applications in iaas clouds. In *Proc. IEEE IPDPS.*
[9] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proc. of IEEE Data Engineering Workshops (ICDEW).*
[10] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. ACM Symposium on Cloud Computing.*
[11] Luyu Li, Houxiang Ji, Chentao Wu, Jie Li, and Minyi Guo. 2017. Favorable Block First: A Comprehensive Cache Scheme to Accelerate Partial Stripe Recovery of Triple Disk Failure Tolerant Arrays. In *Proc. IEEE ICPP.*
[12] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proc. of the 12th ACM International Conf. on Computing Frontiers.*
[13] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems journal* 9, 2 (1970), 78–117.
[14] Russell Power and Jinyang Li. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables.. In *Proc. USENIX OSDI.*
[15] Anna Pupykina and Giovanni Agosta. 2017. Optimizing Memory Management in Deeply Heterogeneous HPC Accelerators. In *Proc. IEEE ICPP Workshops.*
[16] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache tez: A unifying framework for modeling and building data processing applications. In *Proc. ACM SIGMOD.*
[17] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. 2012. M3R: increased performance for in-memory Hadoop jobs. *Proc. the VLDB Endowment* 5, 12 (2012), 1736–1747.
[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proc. of IEEE MSST.*
[19] Apache Storm. http://storm.apache.org/.
[20] SystemG. http://www.cs.vt.edu/facilities/systemg.
[21] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache modeling and optimization using miniature simulations. In *Proc. USENIX ATC.*
[22] Jingjing Wang and Magdalena Balazinska. 2017. Elastic Memory Management for Cloud Data Analytics. In *USENIX ATC.*
[23] Lin Wang. 2013. Directed acyclic graph. In *Encyclopedia of Systems Biology.* Springer, 574–574.
[24] Shaoqi Wang, Xiaobo Zhou, Liqiang Zhang, and Changjun Jiang. 2017. Network-Adaptive Scheduling of Data-Intensive Parallel Jobs with Dependencies in Clusters. In *Proc. of IEEE ICAC.*
[25] Luna Xu, Min Li, Li Zhang, Ali R Butt, Yandong Wang, and Zane Zhenhua Hu. 2016. MEMTUNE: Dynamic memory management for in-memory data analytic platforms. In *In Proc. of IEEE IPDPS.*
[26] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In *Proc. of IEEE INFOCOM.*
[27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of USENIX NSDI.*
[28] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets.. In *Proc. of USENIX HOTCLOUD.*